Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Košir

Prevajalnik za programski jezik C za računalnik SIC/XE

diplomsko delo

univerzitetni strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Prevajalnik za programski jezik C za računalnik SIC/XE

Tematika naloge:

SIC/XE je hipotetični računalnik, ki je bil izdelan kot izobraževalni pripomoček. Računalnik vsebuje vse elemente, ki so potrebni za učinkovito predstavitev osnovnih konceptov področja sistemske programske opreme, hkrati pa je dovolj preprost, da pri njegovi uporabi ne pride do nekaterih težav, ki so značilne za običajne računalnike.

V diplomskem delu predstavite računalnik SIC/XE, programski jezik C ter teoretične pristope za prevajanje višjenivojskega programskega jezika v zbirni jezik. Izdelajte tudi program, ki programsko kodo jezika C prevede v zbirni jezik računalnika SIC/XE. Predstavite preprost program, napisan v programskem jeziku C, in zbirno kodo SIC/XE, ki jo ustvari vaš prevajalnik.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Klemen Košir sem avtor diplomskega dela z naslovom:

Prevajalnik za programski jezik C za računalnik SIC/XE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 1. septembra 2015 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za navdih in pomoč pri diplomskem delu, doc. dr. Boštjanu Slivniku, ki mi je približal svet prevajalnikov, in svojim staršem, ki sta me vse od malih nog podpirala in spodbujala pri izbiri študijske ter življenjske poti. Posebna zahvala gre sestri Tamari za izčrpno poznavanje slovenščine in babici Justini za vso potrpežljivost.

Mojim najdražjim.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SIC/XE	Simplified Instructional Computer / Extra Equipment	razširjena različica preprostega hipotetičnega računalnika
GCC	GNU Compiler Collection	zbirka prevajalnikov za GNU
GNU	GNU's Not Unix!	operacijski sistem GNU
LLVM	Low Level Virtual Machine	nizkonivojski navidezni stroj
ANSI	American National Standards Institute	Ameriški državni inštitut za standarde
ISO	International Organization for Standardization	Mednarodna organizacija za standardizacijo
IEEE	Institute of Electrical and Electronics Engineers	Inštitut inženirjev elektrotehnike in elektronike
NP	nondeterministic polynomial time	nedeterminističen polinomski čas
PC	Program Counter	programski števec
SW	Status Word	stanje računalnika
CC	Condition Code	rezultat primerjave števil
SVC	Supervisor Call	nadzorniški sistemski klic

Povzetek

Cilj tega diplomskega dela je bila izdelava prevajalnika programskega jezika C v zbirni jezik hipotetičnega računalnika SIC/XE. Prevajalnik vsebuje tudi zbirnik, ki iz prevedene zbirne kode ustvari izvedljive objektne datoteke.

Diplomsko delo je razdeljeno na dva dela. Teoretični del na splošno predstavi prevajalnike in programski jezik C ter podrobno opiše arhitekturo računalnikov SIC in SIC/XE. V praktičnem delu smo spoznali strukturo izdelanega prevajalnika in zbirnika ter delovanje posameznih faz. Za analizo izvorne kode programov smo uporabili leksikalna in sintaksna pravila programskega jezika C, za generiranje zbirne kode pa specifikacije računalnika SIC/XE in njegovih objektnih datotek.

V zaključnem delu smo si ogledali težave, s katerimi smo se srečali, in možne izboljšave prevajalnika.

Ključne besede: prevajalniki, programski jezik C, gramatika, računalnik SIC/XE, zbirni jezik, objektne datoteke.

Abstract

The goal of this thesis was to develop a C programming language to assembly language compiler for the hypothetical SIC/XE computer. The compiler also contains an assembler that uses the generated assembly code to create executable object files.

This thesis is separated into two parts. The theoretical part consists of a general overview of compilers and the C programming language, and a detailed description of the SIC and SIC/XE computer architectures. The practical part details the structure of the implemented compiler and assembler, and the function of individual phases. We used lexical and syntactic rules of the C programming language to analyze the source code of the input program, and the specification of the SIC/XE computer and its object files to generate the assembly code.

In the final part of the thesis, we presented the difficulties we encountered, and possible improvements to the compiler.

Keywords: compilers, C programming language, grammar, SIC/XE computer, assembly language, object files.

Poglavje 1

Uvod

V zgodnjem obdobju računalništva so programerji pisali programe v zbirnem jeziku. Stroški razvoja so bili visoki, obenem pa je bilo programiranje na tak način zahtevno in časovno potratno. Zato so bili razviti visokonivojski programski jeziki, ki so programerjem olajšali delo in privabili nove programerje. Računalniki programov, napisanih v visokonivojskih programskih jezikih, ne morejo izvesti, zato je bilo potrebno razviti prevajalnike, ki take programe prevedejo v zbirno oziroma strojno kodo, ki jo računalniki lahko izvedejo.

Cilj tega diplomskega dela je razvoj prevajalnika, ki omogoča prevajanje programov, napisanih v programskem jeziku C, v zbirni jezik hipotetičnega računalnika SIC/XE, in zbirnika, ki ustvari izvedljive objektne datoteke.

V nadaljevanju si bomo ogledali zgodovino in splošno strukturo prevajalnikov ter se seznanili s programskim jezikom C. Nato bomo predstavili arhitekturo in zmožnosti računalnikov SIC in SIC/XE ter podroben opis objektnih datotek, ki s pomočjo nalagalnika omogočajo izvajanje prevedenih programov.

V zadnjem poglavju si bomo podrobno ogledali strukturo implementiranega prevajalnika in delovanje posameznih faz, kar bomo predstavili s pomočjo psevdokode in različnih pravil za razčlenjevanje izvorne kode. Pozornost bomo namenili tudi težavam, s katerimi smo se srečali med izdelavo, in morebitnim rešitvam, ki bi zmanjšale količino generirane zbirne kode.

Poglavje 2

Prevajalniki

Prevajalnik je sistemska programska oprema, ki nam omogoča prevedbo programa, napisanega v izvornem programskem jeziku, v ekvivalentni program, zapisan v ciljnem programskem jeziku. [2]

Najpogosteje jih uporabljamo za prevajanje programov, napisanih v visokonivojskih programskih jezikih, v izvedljive datoteke, ki jih izvede operacijski sistem, ali v objektne datoteke, ki jih s pomočjo nalagalnikov (angl. loader) izvedemo neposredno na računalniku. Poleg tega nam omogočajo tudi prevajanje v zbirno kodo, ki jo programerji lahko uporabijo za razhroščevanje, preizkušanje v simulatorjih in izboljšavo prevajalnikov samih.

Izvorni program je lahko razdeljen na več datotek oziroma modulov, zato mora prevajalnik pred prevajanjem uporabiti predprocesor, s katerim zbere vse potrebne izvorne datoteke in nad njimi izvede makroje, s katerimi v izvorno kodo vstavi konstante ali razširi funkcije. Prevajalnik zatem iz prilagojene izvorne kode generira vmesno kodo, ki vsebuje vse podatke o strukturi izvornega programa in jo optimizira. Vmesna koda se nato prevede v ciljni programski jezik ali zbirno kodo.

Če program prevajamo z namenom, da se izvede na računalniku, prevajalnik generirano zbirno kodo s pomočjo zbirnika (angl. assembler) prevede v objektne datoteke, ki jih nato s povezovalnikom (angl. linker) združi v izvedljivo datoteko. V izvedljivo datoteko vstavi tudi glavo in ustrezne sklice

na funkcije zunanjih knjižnic.

Izvorno kodo programov lahko s posebnimi prevajalniki prevajamo tudi v druge programske jezike, dokler imata izvorni in ciljni programski jezik dovolj podobno strukturo.

2.1 Zgodovina

V zgodnjem obdobju razvoja računalništva so programerji vse programe pisali v zbirnih jezikih, kar je bilo zahtevno in časovno potratno, stroški razvoja pa visoki. Razvoj zmogljivejših računalnikov in visokonivojskih programskih jezikov je programerjem močno olajšal pisanje programov in s tem omogočil razvoj vse bolj kompleksne programske opreme. To pa zahtevalo uporabo prevajalnikov, ki so program, napisan v visokonivojskem programskem jeziku, prevedli nazaj v zbirni jezik oziroma strojno kodo, ki jo računalnik lahko izvede.

Leta 1952 je ameriška računalniška inženirka Grace Murray Hopper napisala prvi prevajalnik in sicer za programski jezik A-0 za računalnik UNIVAC I [5]. Zgodnji prevajalniki so bili napisani v zbirnem jeziku, programerji pa so se neprestano spopadali s pomanjkanjem pomnilnika tedanjih računalnikov in vedno večjo kompleksnostjo samih prevajalnikov.

Napredek v razvoju računalnikov in količini razpoložljivega pomnilnika je olajšal delo programerjev, zato so se razvili prevajalniki, napisani v visokonivojskih programskih jezikih, kasneje pa tudi takšni, ki so bili napisani v programskem jeziku, ki ga prevajajo. Prvi prevajalnik, napisan v visokonivojskem programskem jeziku FLOW-MATIC, je bil prevajalnik za COBOL za računalnik UNIVAC II.

Medtem ko so lahko prvi prevajalniki prevajali en sam programski jezik, so jih kasnejši prevajalniki lahko prevajali več. Obenem so se razvili tudi prevajalniki, ki so programe lahko prevedli za več različnih računalnikov. Struktura takih prevajalnikov je močno olajšala prilagajanje novim programskim jezikom in arhitekturam računalnikov, saj je bilo potrebno napisati le

začelje oziroma zaledje prevajalnika, ki si vmesno kodo in način delovanja deli z obstoječim prevajalnikom.

Vsi večji današnji prevajalniki, kot sta GCC (GNU Compiler Collection) [6] in LLVM (Low Level Virtual Machine), imajo podobno strukturo in so razdeljeni na začelje, srednji del in zaledje. Kljub navidez preprosti strukturi se v ozadju izvaja izjemno zapleten postopek z več prehodi in oblikami vmesne kode, zato razširjanje takih prevajalnikov zahteva podrobno branje dokumentacije in učenje notranjih programskih jezikov, ki se uporabljajo za lažje in hitrejše razčlenjevanje.

2.2 Struktura prevajalnika

Prevajalniki so v splošnem razdeljeni na dva dela:

- začelje (angl. front-end), kjer se izvaja analiza in
- zaledje (angl. back-end), kjer se izvaja sinteza.

V začelju prevajalnik analizira izvorno kodo in tako preveri ustreznost programa ter najde morebitne napake in pomanjkljivosti. Delimo ga na štiri faze:

- **leksikalna analiza:** prevajalnik izvorni program prebere po posameznih znakih in ustvari žetone, ki poenostavijo nadaljnjo analizo,
- **sintaksna analiza:** zaporedje žetonov se primerja z gramatiko izvornega programskega jezika, s čimer preverimo, ali je program napisan v veljavni obliki,
- **semantična analiza:** prevajalnik analizira semantiko izvorne kode (ujemanje imen spremenljivk, podatkovnih tipov ...),
- **generiranje vmesne kode:** iz vseh podatkov, ki smo jih pridobili v prejšnjih fazah, prevajalnik generira vmesno kodo programa.

Ko se izvedejo vse faze začelja, dobimo program, napisan v vmesni kodi. Ta je neodvisna od arhitekture, za katero program prevajamo, kar omogoča izvedbo različnih optimizacij in prevajanje za različne računalnike. Vmesna koda vsebuje vse podatke o izvornem programu, posameznih funkcijah in spremenljivkah ter njihovim podatkovnim tipom.

Oblika vmesne kode ni predpisana in je odvisna od načrtovalca prevajalnika, zato se lahko od prevajalnika do prevajalnika močno razlikuje. Kljub temu je vmesna koda najpogosteje v drevesni obliki, saj tako najlažje predstavimo strukturo izvornega programa.

Prevajalnik lahko vsebuje tudi vmesno fazo, ki analizira generirano vmesno kodo in odstrani vso nedosegljivo kodo, razširi konstante (angl. variable propagation), razvije zanke (angl. loop unrolling) ter izvede druge optimizacije, ki so neodvisne od ciljne arhitekture.

Zaledje prevajalnika vmesno kodo prevede v ciljni programski oziroma zbirni jezik ali strojno kodo. Delimo ga na naslednje faze:

- **optimizacija kode:** izvedejo se različne optimizacije, ki poenostavijo in skrajšajo vmesno kodo (npr. pretvarjanje med različnimi številskimi sistemi oziroma sestavi),

- **sestavljanje klicnih zapisov:** določi se struktura sklada in klicnih zapisov, ki omogočajo gnezdene klice funkcij in shranjevanje vsebine registrov računalnika,

- **generiranje zbirne kode:** prevajalnik generira zbirno kodo, ki namesto registrov uporablja začasne spremenljivke,

- **določanje registrov:** na podlagi analize začasnih spremenljivk in ukazov generirane zbirne kode se spremenljivkam ustrezno določijo registri ciljnega računalnika,

- **optimiziranje generirane kode:** prevajalnik izvede različne optimizacije zbirnih ukazov, ki so odvisne od ciljne arhitekture.

Faze zaledja, ki optimizirajo vmesno oziroma generirano zbirno kodo, se izvedejo v več prehodih, saj lahko določene optimizacije ustvarijo nove priložnosti za dodatno optimizacijo. Optimizacija kode spada med NP-polne (angl. NP-complete) probleme, zato kode programa ne znamo popolnoma optimizirati v polinomskem času.

Če smo izvorni program prevedli z namenom, da se izvede na računalniku (ne prevajamo v drug programski jezik), prevajalnik s pomočjo zbirnika generira strojno kodo in objektne datoteke, nato pa uporabi povezovalnik, s katerim poveže funkcije programa z zunanjimi knjižnicami in ustvari izvedljivo datoteko, ki jo uporabnik lahko izvede na računalniku, za katerega je bil program preveden.

Poglavje 3

Programski jezik C

Programski jezik C je splošno namenski programski jezik, ki ga je Dennis MacAlistair Ritchie leta 1972 razvil za implementacijo operacijskega sistema, prevajalnika in ostalih programov za računalnik DEC PDP-11 [4].

Ker je programski jezik C relativno nizkonivojski, se zelo pogosto uporablja za zamenjavo programov, ki so bili napisani v zbirnem jeziku, saj zagotavlja skoraj enako hitrost izvajanja programov, obenem pa močno olajša programiranje.

Od drugih popularnih programskih jezikov, kot sta Java in C#, se programski jezik C razlikuje v tem, da ne podpira sestavljenih objektov, kot so nizi znakov, množice in seznami, poleg tega pa ne uporablja samodejnega čiščenja pomnilnika (angl. garbage collection). Za slednje morajo skrbeti programerji sami, kar marsikoga odvrne od uporabe programskega jezika C.

Leta 1989 je inštitut ANSI (American National Standards Institute) objavil prvi standard programskega jezika C, ki je poznan pod imenom ANSI C, kasneje pa po letnici objave tudi kot C89. Od leta 1990 namesto inštituta ANSI za standardizacijo skrbi ISO (International Organization for Standardization), ki je do leta 2000 objavil tri nove standarde, ki so prav tako poimenovani po letu objave. Standard C99, ki ga je sprejel tudi ANSI, je še danes najpogosteje uporabljen. Najnovejši standard, imenovan C11 [7], se vse od objave leta 2011 počasi uveljavlja in implementira v prevajalnikih.

Leta 2015 je peta različica prevajalnika GCC privzeti standard C89 nadomestila s standardom C11.

V primerjavi z drugimi popularnimi programskimi jeziki je C nižjenivojski, vendar se kljub temu uporablja za pisanje programov, ki se lahko izvajajo na različnih arhitekturah in platformah (angl. cross-platform). Čeprav C obstaja že od leta 1972 in je v primerjavi z modernimi programskimi jeziki precej omejen, še vedno ostaja eden izmed najpopularnejših.

Uporablja za programiranje tako operacijskih sistemov kot tudi programske opreme za različne arhitekture — od superračunalnikov do vgrajenih sistemov.

Poglavje 4

Hipotetična računalnika SIC in SIC/XE

Hipotetični računalnik SIC je Leland Beck leta 1985 ustvaril za potrebe učenja delovanja sistemske programske opreme v knjigi z naslovom *System Software: An Introduction to Systems Programming* [1]. Računalnik je bil ustvarjen z namenom, da predstavi najpogostejše značilnosti in koncepte strojne opreme, obenem pa se izogiba kompleksnejšim konceptom, ki jih najdemo v obstoječih računalnikih, kot sta registrski sklad in cevovod.

Poleg osnovne različice računalnika SIC poznamo tudi razširjeno različico, imenovano SIC/XE, ki SIC nadgradi z večjim pomnilnikom, dodatnimi registri, novimi načini naslavljanja, podporo za števila s plavajočo vejico in ustreznimi ukazi. SIC/XE je združljiv z računalnikom SIC, zato lahko brez sprememb zbirne kode izvaja programe, ki so bili napisani za SIC.

4.1 Pomnilnik in registri

Računalnika SIC in SIC/XE podpirata pomnilnik velikosti 32768 bajtov oziroma enega megabajta, naslavljata pa posamezne 8-bitne bajte. Iz pomnilnika lahko bereta in zapisujeta 24-bitna predznačena števila, ki se jih naslavlja z najtežjim bitom (angl. big-endian order), ter 8-bitne znake ASCII.

Negativna števila so predstavljena z dvojiškim komplementom.

A	akumulator za aritmetične operacije
X	indeksni register
L	povezavni register
PC	programski števec
SW	statusni register
B	bazni register
S	splošno namenski register
T	splošno namenski register
F	akumulator za operacije s plavajočo vejico

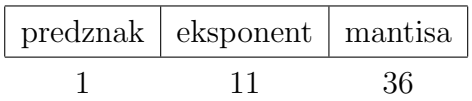
Tabela 4.1: Registri računalnika SIC/XE.

Računalnik SIC ima pet 24-bitnih registrov (A, X, L, PC in SW), ki se uporabljajo za aritmetične operacije s celimi števili, indeksno naslavljanje, klice in vračanje iz podprogramov, sledenje izvajanja ukazov in različne skoke po programu. Poleg teh registrov SIC/XE uporablja še registra za bazno naslavljanje in števila s plavajočo vejico ter dva splošno namenska registra, s katerima lahko izvajamo tudi aritmetične operacije.

4.2 Števila s plavajočo vejico

Računalnik SIC/XE obstoječim 24-bitnim celim številom doda podporo za 48-bitna števila s plavajočo vejico. Zapis teh števil je podoben standardu IEEE 754 [8], razlikuje se le v dolžini posameznih polj bitnega zapisa, kar vpliva na natančnost števil.

SIC/XE uvede tudi nova ukaza, s katerima lahko pretvarjamo med celimi števili in števili s plavajočo vejico.



Slika 4.1: Zapis števila s plavajočo vejico.

Zapis števila s plavajočo vejico sestavljajo:

- 1-bitni predznak: vrednosti 0 in 1 predstavljata negativni oziroma pozitivni predznak,
- 11-bitni eksponent: nepredznačeno število med 0 in 2047,
- 36-bitna mantisa: nepredznačeno decimalno število med 0 in 1,

Število s plavajočo vejico lahko izračunamo z enačbo

$$število = mantisa * 2^{(eksponent - 1024)}$$

Pri tem moramo upoštevati tudi predznak. Število 0 predstavimo tako, da vsem bitom nastavimo vrednost 0.

4.3 Načini naslavljanja

Za izračun naslova, ki se uporabi za dostop do pomnilnika, računalnik SIC uporablja neposredno in indeksno naslavljanje.

Pri neposrednem naslavljanju računalnik SIC kot naslov uporabi nespremenjen odmik, ki je del zbirnega ukaza. Če ima zbirni ukaz nastavljen bit x, računalnik uporabi indeksno naslavljanje, naslovu pa se prišteje vsebina registra X. Indeksno naslavljanje najpogosteje uporabljamo pri zaporednih dostopih do pomnilnika, kot so na primer podatkovna polja in nizi znakov.

Računalnik SIC/XE doda nova dva načina za izračun pomnilniškega naslova. Bazno-relativno naslavljanje za osnovo uporabi vsebino registra B, medtem ko PC-relativno naslavljanje za uporabi programski števec (register PC). Osnovnemu naslovu se nato prišteje še odmik, ki je del zbirnega ukaza.

Odmik pri bazno-relativnem naslavljanju mora biti nepredznačeno število, medtem ko je odmik pri PC-relativnem naslavljanju lahko tudi negativno število, kar omogoča skok na predhodne ukaze. Oba načina izračuna pomnilniškega naslova lahko razširimo tudi z indeksnim naslavljanjem.

Bazno-relativno naslavljanje se uporablja za dostop do višjih pomnilniških naslovov, ki jih sicer ne moremo doseči z 12-bitnim odmikom. PC-relativno

naslavljanje se najpogosteje uporablja v pogojnih ukazih in zankah za skoke po programski kodi.

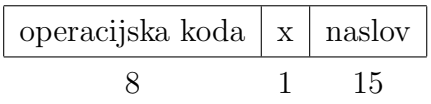
Računalnik SIC/XE lahko izračunani pomnilniški naslov za dostop do operanda uporabi na tri načine:

- z **enostavnim naslavljanjem** (edini način, ki ga podpira SIC), ki operand prebere s pomnilniške lokacije, na katero kaže izračunani naslov,
- s **posrednim naslavljanjem**, ki v pomnilniku z izračunanega naslova prebere nov naslov, ki kaže na pomnilniško lokacijo, kjer se nahaja operand, in
- s **takojšnjim naslavljanjem**, ki ne dostopa do pomnilnika, ampak namesto operanda uporabi uporabi odmik, ki je del zbirnega ukaza.

Takojšnje naslavljanje se uporablja v aritmetičnih ukazih z registrom A, PC-relativne skoke v pogojnih ukazih in nalaganje številskih vrednosti.

4.4 Oblike ukazov

Računalnik SIC za vse ukaze uporablja enako, 24-bitno obliko, kar omogoča hitro in enostavno branje ukazov ter upravljanje s programskim števcem.



Slika 4.2: Oblika ukazov računalnika SIC.

Prvih osem bitov ukaza je operacijska koda, s katero določimo vrsto ukaza, nato pa sledi bit x, ki določi uporabo indeksnega naslavljanja. Zadnjih 15 bitov ukaza vsebuje pomnilniški naslov, na katerem se nahaja operand. Največja vrednost naslova je $2^{15} = 32768$, s katero lahko naslavljamo celoten pomnilnik računalnika SIC. Ta oblika ukazov je zelo omejena, saj ne more vsebovati konstantnih številskih vrednosti. Le-te morajo biti pred izvajanjem programa shranjene v pomnilniku.

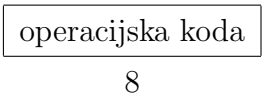
Računalnik SIC/XE nabor ukazov razširi in uporablja štiri različne oblike zbirnih ukazov:

- Prva oblika ukazov je preprosta, saj jo sestavlja le 8-bitna operacijska koda. Ukazi te oblike ne dostopajo do pomnilnika in registrov, uporabljajo pa se za pretvarjanje med celimi števili in števili s plavajočo vejico ter upravljanje z vhodno/izhodnimi kanali.

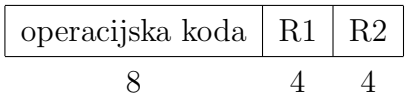
- Drugo obliko uporabljajo ukazi, ki izvajajo aritmetične operacije nad vsebino dveh registrov. Oblika teh ukazov je dolga 16 bitov in je sestavljena iz 8-bitne operacijske kode ter dveh 4-bitnih števil, s katerima naslavljamo posamezne registre.

- Tretja oblika ukazov je na računalniku SIC/XE najpogostejša in je dolga 24 bitov. Sestavljena je iz 6-bitne operacijske kode, šestih dodatnih bitov oziroma zastavic, s katerimi podrobneje določimo delovanje ukaza, in 12-bitnega odmika, ki se uporablja za naslavljanje pomnilnika.

- Četrta oblika je razširjena različica tretje oblike ukazov. Razlikuje se v tem, da namesto 12-bitnega odmika uporablja 20-bitni absolutni naslov, s katerim lahko naslavlja celoten pomnilnik računalnika SIC/XE. Ukazi te oblike so dolgi 40 bitov.



Slika 4.3: Prva oblika ukazov računalnika SIC/XE.



Slika 4.4: Druga oblika ukazov računalnika SIC/XE.

operacijska koda	n	i	x	b	p	e	odmik
6	1	1	1	1	1	1	12

Slika 4.5: Tretja oblika ukazov računalnika SIC/XE.

operacijska koda	n	i	x	b	p	e	naslov
6	1	1	1	1	1	1	12

Slika 4.6: Četrta oblika ukazov računalnika SIC/XE.

Ukazi tretje oblike vsebujejo posebne bite oziroma zastavice, s katerimi lahko določimo, katera načina naslavljanja ukaz uporablja. Razdelimo jih lahko v štiri različne sklope:

- Prvi sklop, sestavljen iz bitov b in p, določa, kako se izračuna pomnilniški naslov, na katerem se nahaja operand. Če je nastavljen bit b, ukaz uporablja bazno-relativno naslavljanje, če pa je nastavljen bit p, uporabimo PC-relativno naslavljanje. V primeru, da ni nastavljen nobeden izmed bitov, ukaz uporabi neposredno naslavljanje.

- Bit x, ki sam sestavlja drugi sklop, določa, ali ukaz pri izračunu pomnilniškega naslova uporablja tudi indeksno naslavljanje.

- Tretji sklop sestavljata bita n in i, s katerima določimo, na kakšen način želimo uporabiti izračunani naslov. V primeru, da sta nastavljena oba bita, ukaz uporablja enostavno naslavljanje. Če je nastavljen bit n, ukaz uporabi posredno naslavljanje, če pa je nastavljen bit i, ukaz uporablja takojšnje naslavljanje.

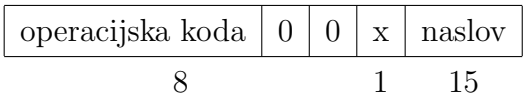
- Zadnji sklop vsebuje le bit e, s katerim določimo, da je ukaz v četrti, razširjeni obliki.

Dolžina ukazov tretje oblike je enaka dolžini ukazov računalnika SIC. Zato jih lahko uporabimo v načinu združljivosti, za kar moramo nastaviti ustrezne bite.

b	bazno-relativno naslavljanje
p	PC-relativno naslavljanje
n	posredno naslavljanje
x	indeksno naslavljanje
i	takojšnje naslavljanje
e	ukaz je v razširjeni obliki

Tabela 4.2: Zastavice ukazov tretje in četrte oblike.

Bita n in i moramo nastaviti na vrednost nič, s čimer določimo, da ukaz uporablja enostavno naslavljanje. Bita tako združimo z operacijsko kodo, saj se bitna oblika vseh operacijskih kod računalnika SIC konča z 00. S tem se dolžina operacijske kode poveča na osem bitov.



Slika 4.7: Oblika ukaza, združljiva z računalnikom SIC.

Ker računalnik SIC ne podpira dodatnih načinov naslavljanja in razširjene oblike ukazov, se biti b, p in e ne uporabljajo, zato jih lahko z odmikom združimo v 15-bitni naslov.

4.5 Ukazi

Računalnik SIC podpira 42 zbirnih ukazov, SIC/XE pa uvede 17 novih, ki se večinoma uporabljajo za aritmetične operacije z registri in števili s plavajočo vejico. Seznam ukazov je prikazan v spodnji tabeli.

ukaz	koda	oblika	delovanje
`ADD m`	18	3/4	A ← A + (m..m+2)
`ADDF m`	58	3/4	F ← F + (m..m+5)
`ADDR r1,r2`	90	2	r2 ← r2 + r1
`AND m`	40	3/4	A ← A & (m..m+2)

`CLEAR r`	B4	2	r ← 0
`COMP m`	28	3/4	CC ← A : (m..m+2)
`COMPF m`	88	3/4	CC ← F : (m..m+5)
`COMPR r1,r2`	A0	2	CC ← r1 : r2
`DIV m`	24	3/4	CC ← A : A / (m..m+2)
`DIVF m`	64	3/4	CC ← F : F / (m..m+5)
`DIVR r1,r2`	9C	2	r2 ← r2 / r1
`FIX`	C4	1	A ← F
`FLOAT`	C0	1	F ← A
`HIO`	F4	1	Zaustavi V/I kanal številka A.
`J m`	3C	3/4	PC ← m
`JEQ m`	30	3/4	PC ← m, če je CC enak =
`JGT m`	34	3/4	PC ← m, če je CC enak >
`JLT m`	38	3/4	PC ← m, če je CC enak <
`JSUB m`	48	3/4	L ← PC; PC ← m
`LDA m`	00	3/4	A ← (m..m+2)
`LDB m`	68	3/4	B ← (m..m+2)
`LDCH m`	50	3/4	A ← (m)
`LDF m`	70	3/4	F ← (m..m+5)
`LDL m`	08	3/4	L ← (m..m+2)
`LDS m`	6C	3/4	S ← (m..m+2)
`LDT m`	74	3/4	T ← (m..m+2)
`LDX m`	04	3/4	X ← (m..m+2)
`LPS m`	D0	3/4	Naloži stanje procesorja.
`MUL m`	20	3/4	A ← A * (m..m+2)
`MULF m`	60	3/4	F ← F * (m..m+5)
`MULR r1,r2`	2	98	r2 ← r2 * r1
`NORM`	C8	1	Normalizira F.
`OR m`	44	3/4	A ← A | (m..m+2)
`RD m`	D8	3/4	A ← DEV[m]
`RMO r1,r2`	AC	2	r2 ← r1

`RSUB`	4C	3/4	PC ← L
`SHIFTL r1,n`	A4	2	r1 ← r1 << n Izvede se krožni pomik.
`SHIFTR r1,n`	A8	2	r1 ← r1 >> n Izbede se aritmetični pomik.
`SIO`	F0	1	Zažene V/I kanal številka A. Naslov programa vsebuje register S.
`SSK m`	EC	3/4	(m) ← A Register A vsebuje zaščitni ključ.
`STA m`	0C	3/4	(m..m+2) ← A
`STB m`	78	3/4	(m..m+2) ← B
`STCH m`	54	3/4	(m) ← A
`STF m`	80	3/4	(m..m+5) ← F
`STI m`	D4	3/4	I ← (m..m+2) Nastavi časovni interval.
`STL m`	14	3/4	(m..m+2) ← L
`STS m`	7C	3/4	(m..m+2) ← S
`STSW m`	E8	3/4	(m..m+2) ← SW
`STT m`	84	3/4	(m..m+2) ← T
`STX m`	10	3/4	(m..m+2) ← X
`SUB m`	1C	3/4	A ← A − (m..m+2)
`SUBF m`	5C	3/4	F ← F − (m..m+5)
`SUBR r1,r2`	94	2	r2 ← r2 − r1
`SVC n`	B0	2	Ustvari prekinitev SVC številka `n`.
`TD m`	E0	3/4	Preveri pripravljenost naprave `m`.
`TIO`	F8	1	Preveri delovanje V/I kanala A.
`TIX m`	2C	3/4	X ← X + 1; CC ← X : (m..m+2)
`TIXR r`	B8	2	X ← X + 1; CC ← X : r
`WD m`	DC	3/4	DEV[m] ← A

Vsi ukazi, ki uporabljajo operande iz pomnilnika, z naslova `m` preberejo tri oziroma šest bajtov.

V zbirnih ukazih lahko uporabimo takojšnje naslavljanje tako, da namesto naslova `m` uporabimo znak `#`, ki mu sledi nepredznačeno število. Če pred naslov `m` dodamo znak `@`, bo ukaz uporabil posredno naslavljanje.

Če pred mnemonik zbirnega ukaza postavimo znak `+`, ukaz uporabi četrto, razširjeno obliko.

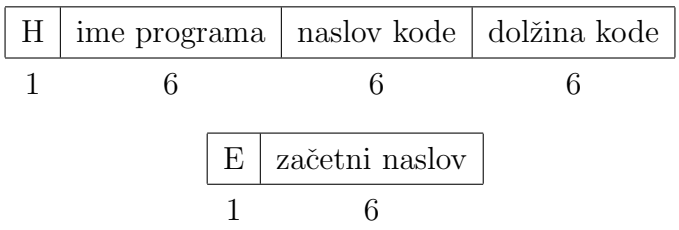
4.6 Objektne datoteke

Zbirnik programe, napisane v zbirni kodi, prevede v objektne datoteke, ki s pomočjo nalagalnika omogočajo izvajanje programov na računalnikih SIC in SIC/XE. Nalagalnik z zapisi objektnih datotek spremenljivkam določi prostor, v pomnilnik naloži strojno kodo programa, nastavi programski števec in začne izvajanje.

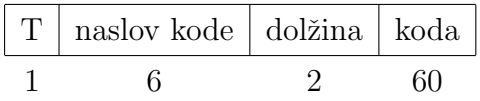
Objektne datoteke so zapisane v besedilni obliki, predstavljene pa so v šestnajstiškem številskem sistemu. Sestavlja jih šest zapisov:

- **zaglavje:** vsebuje ime programa, začetni naslov kode in dolžino strojne kode,
- **programski zapis:** zaporedje strojnih ukazov s skupno dolžino največ 60 bajtov,
- **zapisa za uvoz in izvoz:** določata, katere simbole objektna datoteka uvozi in izvozi v druge objektne datoteke,
- **zapis za prenaslavljanje:** omogoča, da nalagalnik program naloži na poljuben naslov,
- **končni zapis:** označuje konec objektne datoteke in vsebuje naslov, na katerem se začne izvajanje programa.

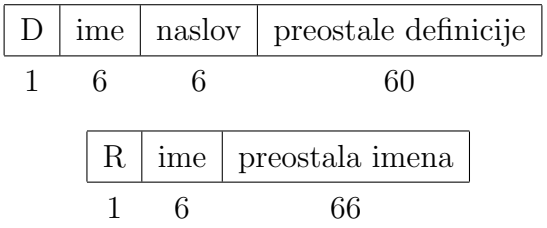
Nalagalnik program naloži tako, da iz prve vrstice objektne datoteke prebere zaglavje, s katerim dobi podatke o velikosti pomnilnika, ki ga program zasede. Zatem prebere posamezne programske zapise in kodo naloži na ustrezne naslove v pomnilniku.



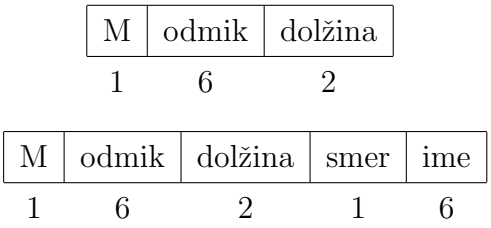
Slika 4.8: Zaglavje in končni zapis objektne datoteke.



Slika 4.9: Programski zapis objektne datoteke.



Slika 4.10: Zapisa za uvoz in izvoz simbolov.



Slika 4.11: Zapisa za prenaslavljanje.

Ko je naložena vsa programska koda, nalagalnik nastavi naslove uvoženih spremenljivk in prenaslovi vse konstantne naslove ukazov. Zatem prebere še končni zapis in začne izvajanje programa na podanem naslovu.

```
HScrSub00000000B853
T0000001AB40053201E4B203D4B20373F2FF40100056D00044B201153200A
T00001A034B2016
T00001D193F2FF73F2FFD2A41000000212025904 00F2FF54F0000072FEF
T000036045790B800
T00003A19B800132FE64F0000532011B4106D27865790B800B8403B2FF7
T000053044F000020
T00B85003030001
M00003705
M00004B05
E000000
```

Izsek kode 4.1: Primer objektne datoteke računalnika SIC/XE.

Poglavje 5

Prevajalnik za SIC/XE

V praktičnem delu bomo predstavili strukturo in delovanje prevajalnika, ki omogoča prevajanje programov, napisanih v programskem jeziku C, v zbirno kodo računalnika SIC/XE, in zbirnika, ki iz zbirne kode ustvari objektne datoteke. Le-te lahko izvedemo v simulatorju računalnika SIC/XE.

Prevajalnik ponuja dve izhodni obliki prevedenega izvornega programa:

- zbirno kodo, namenjeno preizkušanju in iskanju napak ter možnih optimizacij, in
- objektne datoteke, namenjene izključno izvajanju v simulatorju.

Za implementacijo smo uporabili programski jezik Java 8. Ker sta leksikalna in sintaksna analiza del samega prevajalnika, nismo uporabljali namenskih programov, kot sta Flex in Bison.

Zbirnik, ki pretvori zbirno kodo v objektne datoteke, je prav tako napisan v Javi in je kot zadnja faza del samega prevajalnika.

Za potrebe izvajanja in razhroščevanja programov smo uporabili simulator računalnika SIC/XE, ki smo ga med študijem implementirali pri enem izmed predmetov. Napisan je v programskem jeziku C, za prikaz izvajanja pa uporablja knjižnico *ncurses*.

Simulator lahko izvaja tako zbirno kodo kot tudi objektne datoteke, podpira pa prikazovanje vsebine pomnilnika in podrobnosti trenutnega ukaza.

Prevajalnik je razdeljen na tri dele, skupaj pa ga sestavlja deset faz [3]:

- začelje:
  - leksikalna analiza,
  - sintaksna analiza,
  - semantična analiza,
  - sestavljanje klicnih zapisov.
- srednji del:
  - generiranje vmesne kode,
  - linearizacija vmesne kode,
- zaledje:
  - generiranje zbirne kode,
  - določanje registrov,
  - generiranje končne zbirne kode in
  - generiranje objektnih datotek.

V začelju prevajalnika analiziramo vhodni program, preverimo veljavnost in pravilnost izvorne kode ter pripravimo abstraktno sintaksno drevo, ki predstavlja drevesno strukturo izvornega programa. Nato se ustvarijo klicni zapisi, v katerih hranimo stanje registrov in povratne naslove, kar omogoča gnezdenje klicev funkcij.

Srednji del prevajalnika je namenjen generiranju, linearizaciji in optimizaciji vmesne kode, s čimer jo pripravimo na prevajanje v zbirno kodo računalnika SIC/XE.

Zaledje prevajalnika generira zbirno kodo in z analizo začasnih spremenljivk ukazom določi registre. Zatem ustvari datoteko s končno zbirno kodo, ki vsebuje funkcije standardne knjižnice, in objektno datoteko s programom.

V nadaljevanju si bomo podrobneje ogledali posamezne faze prevajalnika. Za prikaz izhoda posameznih faz bomo uporabili program, ki je predstavljen v spodnjem izseku. Namen programa je, da sešteje vsa soda Fibonaccijeva števila, ki so manjša od 4000 [14].

```
int main() {
  int f1 = 1, f2 = 1, result = 0;

  while(f2 < 4000) {
    int f = f1 + f2;
    f1 = f2;
    f2 = f;

    if(f % 2 == 0)
      result = result + f;
  }

  println(result);
  return 0;
}
```

Izsek kode 5.1: Testni program.

5.1 Leksikalna analiza

V fazo leksikalne analize kot vhod podamo datoteko z izvorno kodo v programskem jeziku C. Prevajalnik iz datoteke bere posamezne znake in vsakič preveri, ali se prebrani znak oziroma niz znakov ujema s katerim izmed leksikalnih pravil. Za vsako ujemanje niza znakov prevajalnik ustvari ustrezen žeton (angl. token).

Posamezni žeton sestavljajo podatki o njegovi vrsti, položaju v izvorni datoteki in leksem (angl. lexeme) oziroma njegova znakovna predstavitev. Leksem lahko vsebuje ime spremenljivke, ključno besedo, podatkovni tip ali konstantno vrednost.

Veljavne ključne besede, imena spremenljivk, konstante in drugi znaki so naslednji:

- **ključne besede:** `break, const, continue, do, else, for, if, return, struct, typedef, void, while`,
- **podatkovni tipi:** `boolean, char, float, int`,
- **konstantne vrednosti:**
  - **boolean:** `true, false`,
  - **char:** poljubno število med 32 in 126 (ASCII znaki),
  - **float:** poljubno decimalno število,
  - **int:** poljubno število med -16777216 in 16777215,
  - **short:** poljubno število med -128 in 127.
- **imena:** zaporedje črk, števk in podčrtajev (ime se ne sme začeti s števko in ne sme biti podatkovni tip ali ključna beseda),
- **ostali simboli:** `+ - * / % ++ -- & | && || ^ ! == != < > <= >= << >> ( ) [ ] { } ; . , =`,
- **komentarji:** besedilo od vključno `//` do konca vrstice,
- **belo besedilo:** presledek, tabulator, `\n` in `\r`.

V primeru, da leksem ne ustreza nobenemu izmed leksikalnih pravil, prevajalnik sporoči položaj napake v izvorni datoteki in prekine izvajanje. Izhod leksikalne analize je zaporedje žetonov, ki je strukturno ekvivalentno izvornemu programu.

Spodnji izsek prikazuje primer izhoda leksikalne analize za testni program iz izseka 5.1. Izhod je popolnoma enak vhodnemu programu, vendar posamezne nize znakov nadomestijo žetoni.

```
INT IDENTIFIER(main) LPARENT RPARENT LBRACE
  INT IDENTIFIER(f1) ASSIGN CONST(1) COMMA
      IDENTIFIER(f2) ASSIGN CONST(1) COMMA
      IDENTIFIER(result) ASSIGN CONST(0) SEMICOLON

  WHILE LPARENT IDENTIFIER(f2) LTH CONST(4000) RPARENT LBRACE
    INT IDENTIFIER(f) ASSIGN IDENTIFIER(f1) ADD
        IDENTIFIER(f2) SEMICOLON
    IDENTIFIER(f1) ASSIGN IDENTIFIER(f2) SEMICOLON
    IDENTIFIER(f2) ASSIGN IDENTIFIER(f) SEMICOLON

    IF LPARENT IDENTIFIER(f) MOD CONST(2) EQU CONST(0)
       RPARENT
      IDENTIFIER(result) ASSIGN IDENTIFIER(result) ADD
         IDENTIFIER(f) SEMICOLON
  RBRACE

  IDENTIFIER(println) LPARENT IDENTIFIER(result) RPARENT
     SEMICOLON
  RETURN CONST(0) SEMICOLON
RBRACE
```

Izsek kode 5.2: Izhod leksikalne analize testnega programa.

5.2 Sintaksna analiza

V sintaksni analizi s primerjavo žetonov z gramatiko programskega jezika C preverimo, ali je izvorni program pravilno napisan. Prevajalnik zaporedoma bere žetone, ki jih generira leksikalna analiza, medtem pa v gramatiki išče pravilo, ki ustreza izbranemu zaporedju žetonov.

Prevajalnik zaporedje žetonov preiskuje z razčlenjevalnikom z rekurzivnim poglabljanjem (angl. recursive descend parser) [10]. Medtem izpisuje prepoznana gramatična pravila, obenem pa ustvari abstraktno sintaksno drevo, ki je po obliki enako gramatiki in predstavlja strukturo izvornega pro-

grama. Prevajalnik abstraktno sintaksno drevo uporablja v vseh preostalih fazah začelja.

Izhod sintaksne analize je zaporedje prepoznanih gramatičnih pravil in abstraktno sintaksno drevo.

```
iteration_statement -> while(expression) statement
expression -> logical_or_expression
logical_or_expression -> logical_and_expression
logical_and_expression -> bitwise_or_expression
bitwise_or_expression -> bitwise_xor_expression
bitwise_xor_expression -> bitwise_and_expression
bitwise_and_expression -> equality_expression
equality_expression -> relational_expression
relational_expression -> shift_expression
   relational_expression'
shift_expression -> additive_expression
additive_expression -> multiplicative_expression
multiplicative_expression -> unary_expression
unary_expression -> postfix_expression
postfix_expression -> primary_expression
primary_expression -> identifier
relational_expression' -> < shift_expression
shift_expression -> additive_expression
additive_expression -> multiplicative_expression
multiplicative_expression -> unary_expression
unary_expression -> postfix_expression
postfix_expression -> primary_expression
primary_expression -> int_constant
statement -> compound_statement
...

While 7:2-17:2:
  BinExpr 7:8-7:16: LTH
    Identifier 7:8-7:9: f2
    Constant 7:13-7:16: INTEGER(2000)
```

Izsek kode 5.3: Del izhoda sintaksne analize testnega programa.

V primeru, da za izbrano zaporedje žetonov ne obstaja ustrezno pravilo, prevajalnik programerja obvesti, da program ni veljaven, in prekine prevajanje.

Razčlenjevalnik z rekurzivnim poglabljanjem deluje tako, da za vsako izmed gramatičnih pravil uporablja eno funkcijo. Vsaka izmed funkcij preveri, ali zaporedje žetonov ustreza izbranemu pravilu. V primeru, da funkcija prepozna žeton, ki ustreza enem izmed drugih pravil, pokliče ustrezno funkcijo.

Izsek kode 5.4 prikazuje funkcijo, ki razčlenjuje pravilo za definicijo spremenljivke.

```
function parse_variable_definition()
  print("definition -> variable_definition")
  type = parse_type()
  check(Token.IDENTIFIER)
  expression = null

  token = read_next()

  if token == Token.SEMICOLON
    print("variable_definition -> [type] identifier;")
  else if token == Token.ASSIGN
    print("variable_definition -> [type] identifier =
       expression";)
    expression = parse_expression()
    check(Token.SEMICOLON)
  else
    error("Not a variable definition.")
  endif

  return VARIABLE(type, expression)
```

Izsek kode 5.4: Psevdokoda za razčlenjevanje definicije spremenljivke.

Ker je programski jezik C kompleksen, računalnik SIC/XE pa omejen, smo se odločili, da podpremo le podmnožico vseh konstruktov C-ja in s tem ustrezno zmanjšamo gramatiko sintaksne analize. Za lažje programiranje

smo dodali logični podatkovni tip `boolean`.

Gramatika programskega jezika C [9], ki jo uporablja prevajalnik, je zapisana v obliki Backus-Naur [11]:

⟨*source*⟩ ::= ⟨*definitions*⟩

⟨*definitions*⟩ ::= ⟨*definition*⟩
| ⟨*definition*⟩ `;` ⟨*definitions*⟩

⟨*definition*⟩ ::= ⟨*type-definition*⟩
| ⟨*variable-definition*⟩
| ⟨*function-definition*⟩

⟨*type-definition*⟩ ::= `typedef` ⟨*type*⟩ ⟨*identifier*⟩

⟨*type*⟩ ::= ⟨*identifier*⟩
| `boolean`
| `char`
| `float`
| `int`
| `void`
| `struct` ⟨*identifier*⟩ `{` ⟨*struct-declaration-list*⟩ `}`
| `const` ⟨*type*⟩
| `*`⟨*type*⟩

⟨*struct-declaration-list*⟩ ::= ⟨*struct-declaration*⟩
| ⟨*struct-declaration*⟩ ⟨*struct-declaration-list*⟩

⟨*struct-declaration*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩`;`

⟨*variable-definition*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩
| ⟨*type*⟩ ⟨*identifier*⟩`[`⟨*constant*⟩`]`
| ⟨*type*⟩ ⟨*identifier*⟩ `=` ⟨*expression*⟩

⟨*function-definition*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩`(`⟨*parameters*⟩`)`
⟨*compound-statement*⟩

⟨*parameters*⟩ ::= ⟨*parameter*⟩
| ⟨*parameter*⟩`,` ⟨*parameters*⟩

⟨*parameter*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩

⟨*compount-statement*⟩ ::= `{` ⟨*block-items*⟩ `}`
| `{ }`

⟨*block-items*⟩ ::= ⟨*block-item*⟩
| ⟨*block-item*⟩ ⟨*block-items*⟩

⟨*block-item*⟩ ::= ⟨*variable-definition*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*selection-statement*⟩
| ⟨*iteration-statement*⟩
| ⟨*jump-statement*⟩
| ⟨*expression-statement*⟩
| ⟨*compound-statement*⟩

⟨*selection-statement*⟩ ::= `if(`⟨*expression*⟩`)` ⟨*statement*⟩
| `if(`⟨*expression*⟩`)` ⟨*statement*⟩ `else` ⟨*statement*⟩

⟨*iteration-statement*⟩ ::= `while(`⟨*expression*⟩`)` ⟨*statement*⟩
| `do` ⟨*statement*⟩ `while(`⟨*expression*⟩`);`
| `for(`⟨*variable-definition*⟩ ⟨*expression-statement*⟩`)` ⟨*statement*⟩
| `for(`⟨*variable-definition*⟩ ⟨*expression-statement*⟩ ⟨*expression*⟩`)`
⟨*statement*⟩
| `for(`⟨*expression*⟩ ⟨*expression-statement*⟩`)` ⟨*statement*⟩
| `for(`⟨*expression*⟩ ⟨*expression-statement*⟩ ⟨*expression*⟩`)` ⟨*statement*⟩

⟨*jump-statement*⟩ ::= `break;`
| `continue;`
| `return;`
| `return` ⟨*expression*⟩`;`

⟨*expression-statement*⟩ ::= ⟨*expression*⟩ `;`
| `;`

⟨*expression*⟩ ::= ⟨*logical-or-expression*⟩
| ⟨*logical-or-expression*⟩ `=` ⟨*expression*⟩

⟨*logical-or-expression*⟩ ::= ⟨*logical-and-expression*⟩
| ⟨*logical-or-expression*⟩ `||` ⟨*logical-and-expression*⟩

⟨*logical-and-expression*⟩ ::= ⟨*bitwise-or-expression*⟩
| ⟨*logical-and-expression*⟩ `&&` ⟨*bitwise-or-expression*⟩

⟨*bitwise-or-expression*⟩ ::= ⟨*bitwise-and-expression*⟩
| ⟨*bitwise-or-expression*⟩ `|` ⟨*bitwise-and-expression*⟩

⟨*bitwise-and-expression*⟩ ::= ⟨*equality-expression*⟩
| ⟨*bitwise-and-expression*⟩ `&` ⟨*equality-expression*⟩

⟨*equality-expression*⟩ ::= ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ `==` ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ `!=` ⟨*relational-expression*⟩

⟨*relational-expression*⟩ ::= ⟨*shift-expression*⟩
| ⟨*relational-expression*⟩ `<` ⟨*shift-expression*⟩
| ⟨*relational-expression*⟩ `>` ⟨*shift-expression*⟩
| ⟨*relational-expression*⟩ `<=` ⟨*shift-expression*⟩
| ⟨*relational-expression*⟩ `>=` ⟨*shift-expression*⟩

⟨*shift-expression*⟩ ::= ⟨*additive-expression*⟩
| ⟨*shift-expression*⟩ `<<` ⟨*additive-expression*⟩
| ⟨*shift-expression*⟩ `>>` ⟨*additive-expression*⟩

⟨*additive-expression*⟩ ::= ⟨*multiplicative-expression*⟩
| ⟨*additive-expression*⟩ `+` ⟨*multiplicative-expression*⟩
| ⟨*additive-expression*⟩ `-` ⟨*multiplicative-expression*⟩

⟨*multiplicative-expression*⟩ ::= ⟨*unary-expression*⟩
| ⟨*multiplicative-expression*⟩ `*` ⟨*unary-expression*⟩
| ⟨*multiplicative-expression*⟩ `/` ⟨*unary-expression*⟩
| ⟨*multiplicative-expression*⟩ `%` ⟨*unary-expression*⟩

⟨*unary-expression*⟩ ::= ⟨*postfix-expression*⟩
| `!`⟨*unary-expression*⟩
| `-`⟨*unary-expression*⟩
| `++`⟨*unary-expression*⟩
| `--`⟨*unary-expression*⟩

⟨*postfix-expression*⟩ ::= ⟨*primary-expression*⟩
| ⟨*primary-expression*⟩`++`
| ⟨*primary-expression*⟩`--`
| ⟨*postfix-expression*⟩`.`⟨*identifier*⟩
| ⟨*postfix-expression*⟩`[`⟨*expression*⟩`]`

⟨*primary-expression*⟩ ::= `boolean_constant`
| `integer_constant`
| `float_constant`
| `character_constant`
| `string_constant`
| ⟨*identifier*⟩
| ⟨*identifier*⟩`(`⟨*arguments*⟩`)`
| `(`⟨*expression*⟩`)`

⟨*identifier*⟩ ::= `name`
| `*`⟨*identifier*⟩

⟨*arguments*⟩ ::= ⟨*expression*⟩
| ⟨*expression*⟩`,` ⟨*arguments*⟩

5.3 Semantična analiza

Semantična analiza kot vhod sprejme abstraktno sintaksno drevo, ki trenutno vsebuje le podatke o strukturi izvornega programa. V tej fazi se obstoječim podatkom dodajo povezave do definicij spremenljivk in funkcij ter podrobnosti o podatkovnih tipih posameznih spremenljivk, konstant in ostalih konstruktov izvornega programa.

```
function analyze(node)
  analyze(node.left_expression)
  analyze(node.right_expression)

  if node.operation == (ADD | SUB | MUL | DIV)
    if node.left_expression == node.right_expression
      set(node, node.left_expression)
    else
      report(ERROR)
    endif
  endif

  if node.operation == (OR | AND)
    if node.left_expression == (INTEGER | BOOLEAN) &
       node.right_expression == (INTEGER | BOOLEAN)
      set(node, BOOLEAN)
    else
      report(ERROR)
    endif
  endif

  ...
```

Izsek kode 5.5: Del psevdokode za semantično analizo operacij z dvema operandoma.

Prevajalnik na začetku semantične analize s preiskovanjem v globino obišče vsako vozlišče drevesa in za vsak uporabljen podatkovni tip, funkcijo ali spremenljivko preveri, ali je v kodi že definirana. S tem ugotovimo,

ali se v programu pojavi dostop do nedefinirane spremenljivke, obenem pa lahko programerja obvestimo, če obstajajo neuporabljene spremenljivke.

Poleg povezovanja dostopov do spremenljivk in funkcij z njihovimi definicijami prevajalnik preveri tudi ujemanje podatkovnih tipov. Preveriti je potrebno vse konstante, dodeljevanje spremenljivk, podatkovne tipe v aritmetičnih operacijah in podatkovne tipe vseh drugih izrazov.

Prevajalnik med semantično analizo za preiskovanje drevesa uporablja rekurzivno poglabljanje. V vsakem vozlišču najprej obišče podrejena vozlišča in določi njihove podatkovne tipe. Nato glede na vrsto izraza preveri ustreznost podatkovnih tipov podizrazov, določi podatkovni tip vozlišča in se vrne v nadrejeno vozlišče.

Podatke o vrsti in velikosti podatkovnih tipov prav tako dodamo v abstraktno sintaksno drevo, saj jih potrebujemo tudi v nadaljnjih fazah za izračun ustreznih naslovov.

5.4 Sestavljanje klicnih zapisov

V zadnji fazi začelja prevajalnik pripravi strukturo klicnih zapisov, ki se uporabljajo za shranjevanje lokalnih spremenljivk in stanja registrov. S tem klicanim funkcijam omogočimo dostop do vseh registrov, obenem pa zagotovimo, da lokalne spremenljivke ne prepišejo spremenljivk klicoče funkcije.

Klicni zapisi funkcij se nahajajo na skladu v pomnilniku. Za upravljanje s klicnim zapisom skrbi vsaka funkcija sama.

FP	lokalne spremenljivke
	stara vrednost FP
	povratni naslov
	začasne spremenljivke
	shranjeni registri
SP	argumenti funkcij

Tabela 5.1: Klicni zapis funkcije.

Klicni zapis posamezne funkcije vsebuje šest podatkovnih polj:

- **lokalne spremenljivke:** vsebuje vse spremenljivke, ki so definirane znotraj funkcije,
- **staro vrednost FP:** kazalec, ki kaže na začetek prejšnjega klicnega zapisa in se uporablja za odstranjevanje starih klicnih zapisov s sklada,
- **povratni naslov:** naslov, na katerega se funkcija vrne po koncu izvajanja, ki omogoča gnezdene klice,
- **začasne spremenljivke:** v primeru, da spremenljivkam zmanjka registrov, se vsebina registrov, ki jih trenutno ne uporabljamo, shrani v to polje,
- **shranjene registre:** pred začetkom izvajanja funkcije se v to polje shrani vsebina vseh registrov, s čimer zagotovimo nespremenjeno stanje klicoče funkcije, in
- **argumente funkcij:** vanj shranimo argumente klicanih funkcij.

Za dostop do klicnih zapisov na skladu potrebujemo tudi dva kazalca: kazalec FP (angl. Frame Pointer), ki kaže na zadnji klicni zapis na skladu oziroma klicni zapis trenutne funkcije, in kazalec SP (angl. Stack Pointer), ki kaže na vrh sklada.

Pred vsakim klicem funkcije klicoča funkcija v klicni zapis shrani argumente klicane funkcije. Klicana funkcija nato na vrhu sklada ustvari nov klicni zapis in ustrezno premakne kazalca FP in SP. Tako kazalec FP kaže na nov klicni zapis, kazalec SP pa na nov vrh sklada.

Med izvajanjem funkcije se argumenti ne nahajajo v klicnem zapisu trenutne funkcije, ampak v podatkovnem polju klicoče funkcije.

Ko se izvajanje trenutne funkcije zaključi, se kazalec SP premakne na konec prejšnjega klicnega zapisa, kazalec FP iz klicnega zapisa prebere staro vrednost. S tem se klicni zapis klicane funkcije odstrani s sklada.

Prevajalnik za vsako funkcijo izračuna velikost klicnega zapisa oziroma posameznih podatkovnih polj. V tej fazi je mogoče izračunati le velikost lokalnih spremenljivk in argumentov klicanih funkcij ter povratnega naslova in stare vrednosti kazalca FP. Velikost začasnih spremenljivk in shranjenih registrov se lahko izračuna šele v predzadnji fazi prevajanja, saj prevajalnik trenutno ne ve, koliko registrov bo funkcija potrebovala med izvajanjem.

Izračun velikosti podatkovnih polj je enostaven. Velikosti lokalnih spremenljivk izračunamo tako, da v funkciji poiščemo vse definicije spremenljivk in seštejemo njihove velikosti, medtem ko sta povratni naslov in stara vrednost kazalca FP konstantne velikosti treh bajtov, saj predstavljata naslov v pomnilniku računalnika SIC/XE.

Velikost polja z argumenti klicanih funkcij prevajalnik izračuna tako, da v trenutni funkciji poišče klic funkcije, katere argumenti zasedejo največ prostora. S tem zagotovi, da ima klicni zapis dovolj prostora za argumente vseh klicanih funkcij. Argumenti v klicnem zapisu se ob vsakem klicu prepišejo, klicana funkcija pa dostopa le do argumentov, ki jih potrebuje.

5.5 Generiranje vmesne kode

Program, zapisan v abstraktnem sintaksnem drevesu, se v srednjem delu prevajalnika pretvori v vmesno kodo, ki je neodvisna od ciljne arhitekture. Vmesna koda ohranja drevesno strukturo, obenem pa se približa obliki zbirne kode računalnika SIC/XE.

Drevesno vmesno kodo sestavljajo naslednji elementi:

- izrazi:
  - CONST: konstanta številska vrednost ali niz znakov,
  - GLOBAL: naslov globalne spremenljivke,
  - TEMP: začasna spremenljivka,
  - BINOP: izraz z dvema operandoma,
  - CALL: klic funkcije,

  - MEM: branje z naslova v pomnilniku.

- stavki:

  - MOVE: premik ene začasne spremenljivke v drugo,
  - LABEL: oznaka,
  - JUMP: skok na oznako,
  - CJUMP: pogojni skok na oznako,
  - SEQ: zaporedje stavkov in izrazov.

Tako kot v prejšnjih fazah prevajalnik tudi v tej rekurzivno obišče vsako vozlišče abstraktnega sintaksnega drevesa in generira vmesno kodo.

V tej fazi se ne generira ena sama vmesna drevesna koda za celotne program, ampak ima vsaka izmed funkcije ločeno vmesno kodo. Prav tako se generira vmesna koda globalnih spremenljivk, ki vsebuje ime oziroma naslov spremenljivke in velikost ter morebitni izraz, ki določa začetno vrednost oziroma vsebino spremenljivke.

```
function generate(node)
  generate(node.condition)
  generate(node.true)
  generate(node.false)

  var sequence = SEQ(
    CJUMP(node.condition, true_label, false_label),
    LABEL(true_label),
    SEQ(node.true),
    LABEL(false_label),
    SEQ(node.false)
  )

  set(node, sequence)
```

Izsek kode 5.6: Psevdokoda za generiranje vmesne kode pogojnega stavka *if*.

```
SEQ
  CJUMP(true_label, false_label)
    BINOP(==)
      BINOP(%)
        MEM
          MOVE
            TEMP(f)
            MEM
              BINOP(+)
                TEMP(FP)
                TEMP(0)
        CONST(2)
      CONST(0)
  LABEL(true_label)
  SEQ
    MOVE
      TEMP(result)
      BINOP(+)
        TEMP(FP)
        CONST(-3)
    MOVE
      TEMP(f)
      MEM
        BINOP(+)
          TEMP(FP)
          CONST(0)
    MOVE
      TEMP(result)
      BINOP(+)
        MEM
          TEMP(result)
        TEMP(f)
  LABEL(false_label)
```

Izsek kode 5.7: Del vmesne kode testnega programa.

5.6 Linearizacija vmesne kode

Preden prevajalnik generira zbirno kodo za računalnik SIC/XE, je potrebno linearizirati vmesno drevesno kodo, s čimer poenostavimo in zmanjšamo globino dreves ter jih razdelimo v več manjših. S tem omogočimo lažje generiranje zbirne kode in izvedbo optimizacij, kot je odstranjevanje nedosegljive kode in nepotrebnih aritmetičnih operacij.

Glavni cilj linearizacije je, da iz drevesa odstranimo vse vgnezdene stavke SEQ, kar pomeni, da drevo posamezne funkcije vsebuje eno samo zaporedje izrazov. S tem se oblika vmesne kode že zelo približa obliki zbirne kode.

```
MOVE
  TEMP(f)
  MEM
    TEMP(FP)
CJUMP(true_label, false_label)
  BINOP(==)
    BINOP(%)
      TEMP(f)
      CONST(2)
    CONST(0)
  LABEL(true_label)
  MOVE
    TEMP(result)
    BINOP(+)
      TEMP(FP)
      CONST(-3)
  MOVE
    TEMP(result)
    BINOP(+)
      MEM
        TEMP(result)
          TEMP(f)
  LABEL(false_label)
```

Izsek kode 5.8: Del linearizirane vmesne kode testnega programa.

Izsek 5.8 prikazuje vmesno kodo iz izseka kode 5.7 po linearizaciji. Odstranjena so zaporedja stavkov SEQ, nepotrebno prištevanje vrednosti 0 in odvečna branja iz pomnilnika.

Linearizacija kode prav tako odstrani klice funkcij znotraj vgnezdenih izrazov. Klici funkcij se po linearizaciji izvedejo pred izvirnim izrazom, rezultat pa se shrani v začasno spremenljivko. Le-ta nadomesti klic funkcije znotraj vgnezdenega izraza.

5.7 Generiranje zbirne kode

Takoj po izvedbi linearizacije vmesne kode se začne zaledje prevajalnika in faza generiranja zbirne kode za računalnik SIC/XE.

Generiranje zbirne kode se izvede v dveh prehodih:

- prevajalnik za vsako vozlišče vmesne drevesne kode generira zbirne ukaze, in
- optimizira generirano kodo.

V postopku generiranja zbirne kode prevajalnik zaporedno obišče drevesa linearizirane vmesne kode posameznih funkcij in za vsako vozlišče generira zaporedje zbirnih ukazov. Le-ti trenutno ne uporabljajo registrov računalnika SIC/XE, ampak začasne spremenljivke.

Prevajalnik nad generirano zbirno kodo izvede dve različni optimizaciji. Prva optimizacija poskrbi, da zbirna koda ne vsebuje več zaporednih oznak in oznak, ki se nahajajo takoj pred ukazom, ki izvede brezpogojni skok na novo oznako. Z drugo, manjšo optimizacijo pa prevajalnik odstrani vse zbirne ukaze oblike `RMO R,R`, ki vsebino registra R prestavi v isti register.

Primer psevdokode za generiranje zbirnih ukazov za nalaganje številskih vrednosti je predstavljen v izseku 5.9.

Pri generiranju zbirne kode smo se srečali z več težavami, katerih vzrok je večinoma omejenost računalnika SIC/XE.

```
function generate(node)
  if node.value < 0
    var constant = abs(node.value)
    code = {
      "CLEAR [node.variable]",
      "LD[temp] #[node.constant]",
      "SUBR [temp], [node.variable]"
    }
  else
    code = "LD[node.variable] #[node.value]"
  endif

  set(node, code)
```

Izsek kode 5.9: Del psevdokode za generiranje zbirne kode za nalaganje števil.

Največja težava, ki smo jo opazili, je pomanjkanje aritmetičnih zbirnih ukazov, ki kot operand uporabljajo register in številsko vrednost ter ukaza za branje iz pomnilnika, če se pomnilniški naslov nahaja v registru.

Če želimo iz klicnega zapisa prebrati vrednost lokalne spremenljivke, ki se nahaja na naslovu FP-3, moramo izvesti sedem ukazov:

1. `CLEAR B`: vsebino registra B nastavimo na vrednost 0,
2. `LDA #3`: v register A naložimo vrednost 3,
3. `SUBR A,B`: vsebino registra A odštejemo od registra B,
4. `LDA FP`: v register A naložimo vrednost kazalca FP,
5. `ADDR B,A`: vsebino registra B prištejemo registru A,
6. `STA ADDRESS`: izračunani naslov v registru A shranimo v pomnilnik na naslov oznake ADDRESS,
7. `LDA @ADDRESS`: s posrednim naslavljanjem preberemo vsebino pomnilnika na naslovu FP-3.

Število zbirnih ukazov generirane kode bi lahko zmanjšali, če bi računalnik SIC/XE podpiral dodatno obliko ukazov, ki je podobna drugi obliki. Ta oblika bi imela obrnjeno zaporedje operandov, register `r2` pa bi nadomestila številska vrednost.

Tako bi lahko uporabili ukaze, kot je `r1 = r1 + #`. Posledično bi lahko prvih pet ukazov nadomestili z ukazoma `LDA FP` in `SUBR A,#3`.

Poleg tega bi računalniku SIC/XE lahko dodali nov način naslavljanja, s katerim bi do pomnilnika dostopal preko naslova, ki je shranjen v registru. Ker za vsak dostop do pomnilnika (razen pri dostopu do globalnih spremenljivk) dostopamo preko kazalca FP, moramo vsakič izračunati ciljni naslov, ga shraniti v pomnilnik in nato s posrednim naslavljanjem dostopati do želene vrednosti. Če bi SIC/XE podpiral posredno naslavljanje preko registrov, bi se lahko znebili nepotrebnega dostopa do pomnilnika in zadnja dva ukaza nadomestili z ukazom `LDA @A`.

Tako bi za dostop do spremenljivke potrebovali le tri ukaze:

1. `LDA FP`: v register A naložimo vrednost kazalca FP,
2. `SUBR A,#3`: od registra A odštejemo število 3,
3. `LDA @A`: s posrednim naslavljanjem preberemo vrednost z naslova FP-3.

S tem bi prav tako zmanjšali število uporabljenih registrov, obenem pa povečali zmogljivost optimizacij, saj bi odstranili veliko večino zbirnih ukazov, ki nalagajo številske vrednosti.

Naslednja težava, s katero smo se srečali, je dejstvo, da SIC/XE ne podpira operatorja `%`, s katerim izračunamo ostanek pri deljenju. Zato smo morali vsako tako operacijo nadomestiti z odštevanjem, množenjem, deljenjem in dodatnimi dostopi do pomnilnika.

Zadnja, sicer manjša težava je, da računalnik SIC/XE nima ukaza `JNE`, ki bi preveril, ali je rezultat zadnje primerjave različen od nič. Zato moramo pri vsaki taki primerjavi z ukazoma `JGT` in `JLT` preveriti, ali je rezultat večji ali manjši od nič.

5.8 Določanje registrov

Ko prevajalnik generira zbirno kodo, je potrebno vse začasne spremenljivke nadomestiti z registri računalnika SIC/XE. Prevajalnik najprej analizira zaporedje ukazov in začasnih spremenljivk ter generira interferenčni graf, na podlagi katerega lahko ukazom določi registre.

Pri analiziranju začasnih spremenljivk prevajalnik uporabi sezname definiranih in uporabljenih spremenljivk posameznega zbirnega ukaza, ki smo jih določili pri generiranju zbirne kode. Prevajalnik s pomočjo algoritma za analizo spremenljivk za vsak ukaz ustvari seznam spremenljivk, ki se dejavne v času izvajanja ukaza. S tem lahko določimo življenjsko dobo posameznih začasnih spremenljivk in število registrov, ki jih računalnik SIX/XE potrebuje v času izvajanja posameznega ukaza.

```
do {
  for instr in reverse(instructions) {
    instr.in_old = instr.in
    instr.out_old = instr.out

    instr.in = instr.uses ∪ (instr.out \ instr.defs)
    instr.out = ∪ instr.successors.in
  }
} until ∀instr:
  instr.in == instr.in_old and instr.out == instr.out_old
```

Izsek kode 5.10: Psevdokoda za analizo začasnih spremenljivk.

Algoritem deluje tako, da obratnem vrstnem redu zaporedno obišče vsak zbirni ukaz in glede na predhodne ter naslednje ukaze določi spremenljivke, ki vstopijo in izstopijo iz ukaza. To ponavlja, dokler so spremembe.

Iz seznama spremenljivk prevajalnik ustvari interferenčni graf, ki prikazuje, med katerimi začasnimi spremenljivkami pride do interference. Interferenca med dvema začasnima spremenljivkama pomeni, da sta dejavni v istem zbirnem ukazu, zato jima ne moremo določiti istega registra.

Registri se zbirnim ukazom določijo z algoritmom za barvanje interfe-

renčnega grafa, kjer vsaka barva predstavlja enega izmed registrov. Cilj algoritma je, da vsako začasno spremenljivko v interferenčnem grafu pobarva z eno barvo, vendar morajo biti spremenljivke, med katerimi pride do interference, različnih barv. S tem zagotovimo, da se spremenljivke, ki so dejavne v istem zbirnem ukazu, nahajajo v različnih registrih računalnika SIC/XE.

Ker je barvanje grafa NP-težek problem, za barvanje ne uporabljamo algoritma, ki poišče najboljšo rešitev, ampak samo približek.

Algoritem za barvanje poleg interferenčnega grafa [12], [13] potrebuje sklad, na katerega začasno premakne posamezna vozlišča, in vrednost R, s katero določimo, s koliko registri računalnik razpolaga. V primeru računalnika SIC/XE je vrednost R enaka 4. Algoritem je sestavljen iz petih korakov:

1. **build:** z analizo začasnih spremenljivk sestavimo interferenčni graf,

2. **simplify:** vsako vozlišče, ki ima manj kot R povezav, odstranimo iz grafa in ga premaknemo na sklad,

3. **spill:** če v grafu obstaja vozlišče z R ali več povezavami, ga označimo kot morebitni preliv in premaknemo na sklad ter se vrnemo na prejšnji korak,

4. **select:** vozlišča s sklada vračamo v graf in jih sproti barvamo; če morebitnega preliva ni mogoče pobarvati, ga razglasimo za dejanski preliv in nadaljujemo z naslednjim korakom, sicer pa se algoritem konča,

5. **start-over:** vse začasne spremenljivke, ki imajo dejanski preliv, shranimo v klicni zapis in ponovimo celoten algoritem.

Algoritem z drugim korakom zagotovi, da se iz interferenčnega grafa odstranijo vse spremenljivke, za katere zagotovo vemo, da jih lahko pobarvamo.

Prevajalnik v tretjem koraku izbere eno izmed spremenljivk, ki ima več povezav, kot ima računalnik SIC/XE na voljo registrov. To spremenljivko označi kot morebitni preliv, saj se lahko zgodi, da so nekatere izmed sosednjih spremenljivk v grafu istih barv, kar pomeni, da lahko le-to vozlišče še vedno

pobarvamo. Ko morebitni preliv odstranimo iz grafa, se algoritem vrne na drugi korak, saj imajo tedaj nekatere spremenljivke kot R povezav.

Vsaki vrnjeni spremenljivki prevajalnik določi eno izmed barv, s katero ni pobarvana nobena izmed sosednjih spremenljivk. Do izjeme pride pri vračanju spremenljivk z morebitnim prelivom. Vsako tako spremenljivko prevajalnik v primeru, da je ni mogoče pobarvati, razglasi za dejanski preliv in pobarva s posebno barvo.

Vse spremenljivke, pri katerih pride do dejanskega preliva, moramo shraniti v namensko podatkovno polje v klicnem zapisu. Računalnik SIC/XE spremenljivk z dejanskim prelivom ne sme hraniti v registrih, vendar jo mora pred vsako uporabo prebrati iz pomnilnika, nato pa shraniti nazaj. To izvedemo tako, da vsak zbirni ukaz, ki to spremenljivko uporablja, obdamo z novima ukazoma, ki prebereta oziroma shranita spremenljivko v klicni zapis.

Za hitrejše izvajanje zbirne kode in manjše število dostopov do pomnilnika, prevajalnik kazalec FP hrani v enem izmed registrov.

5.9 Generiranje končne zbirne kode

Preden se lahko generirana zbirna koda izvede v simulatorju računalnika SIC/XE, mora prevajalnik dodati še nekaj malenkosti.

Najprej dodamo glavno funkcijo `Start`, v kateri računalnik SIC/XE nastavi vrednost kazalcev FP in SP ter pokliče funkcijo `_main`, ki predstavlja začetek prevedenega programa. Kazalca FP in SP imata na začetku izvajanja enako vrednost in kažeta na zadnji naslov pomnilnika. Funkcija prav tako z ukazom `CLEAR` počisti vsebino vseh registrov in s tem prepreči morebitne napake.

Prevajalnik nato vsaki funkciji doda prolog in epilog, ki predstavljata začetni in končni del posamezne funkcije.

Prolog funkcije najprej shrani vsebino vseh registrov, nato pa premakne kazalca FP in SP, s čimer ustvari nov klicni zapis. Zatem shrani staro vrednost kazalca FP in povratni naslov. S tem zagotovi, da klicana funkcija

ne pokvari vsebine registrov klicoče funkcije. Epilog funkcije shrani rezultat funkcije v klicni zapis, nato pa v obratnem vrstnem redu razveljavi delo prologa in se vrne v klicočo funkcijo.

```
STA TEMP                    STA  TEMP
LDA SP                      LDA  FP
SUB #ODMIK_REG              ADD  #ODMIK_REZ
STA ADDR                    STA  ADDR
LDA TEMP                    LDA  TEMP
STA @ADDR                   STA  @ADDR
LDA ADDR                    LDA  FP
SUB #3                      STA  SP
STA ADDR                    SUB  #ODMIK_PN
STB @ADDR                   STA  ADDR
SUB #3                      LDL  @ADDR
STA ADDR                    SUB  #3
STS @ADDR                   STA  ADDR
...                         LDA  @ADDR
LDA SP                      STA  FP
SUB #ODMIK_PN               SUB  #ODMIK_REG
STA ADDR                    STA  ADDR
STL @ADDR                   ...
SUB #3                      LDB  @ADDR
STA ADDR                    ADD  #3
LDA FP                      STA  ADDR
STA @ADDR                   LDA  @ADDR
LDA SP                      RSUB
STA FP
ADD #DOLŽINA
STA SP
```

Izsek kode 5.11: Zbirna koda prologa in epiloga funkcije.

Zgornji izsek kode prikazuje prolog in epilog funkcije. Odmiki `ODMIK_REG`, `ODMIK_PN` in `ODMIK_REZ` predstavljajo odmike od začetka klicnega zapisa do podatkovnih polj za shranjene registre, povratni naslov in rezultat funkcije. `DOLŽINA` predstavlja dolžino klicnega zapisa. Vse konstante se izračunajo

med prevajanjem.

Na konec prevedenega programa prevajalnik doda še direktive zbirnega jezika računalnika SIC/XE, s katerimi določimo naslove in začetne vrednosti globalnih spremenljivk. Prevajalnik prav tako doda uporabljene funkcije standardne knjižnice, kot je na primer `println()`. Celotna generirana zbirna koda se zapiše v izhodno datoteko.

Spodnji izsek kode prikazuje končno generirano zbirno kodo testnega programa, ki jo lahko izvedemo v simulatorju računalnika SIC/XE.

```
FP      WORD  X'000FFF'     # Kazalca FP in SP.
SP      WORD  X'000FFF'
ADDR    RESW  1             # Začasna spremenljivka
                              za posredno naslavljanje.
Start   START 100
        LDX   FP
        ...                 # Počistimo registre.
        JSUB  _main
End     J     End

_main   ...                 # Prolog funkcije.
        CLEAR S             # int f1 = 1;
        LDA   #3
        SUBR  A,S
        RMO   X,A
        ADDR  S,A
        LDS   #1
        STA   ADDR
        STS   @ADDR
        ...
L0      CLEAR S             # Začetek zanke.
        LDA   #6            # while(f2 < 4000)
        SUBR  A,S
        RMO   X,A
        ADDR  S,A
        STA   ADDR
        LDS   @ADDR         # Preberemo f2.
```

```
        LDA    #4000
        COMPR S,A           # Primerjamo števili.
        JLT    L1
        J      L3
L1      CLEAR S             # Telo zanke.
        LDA    #12          # int f = f1 + f2;
        SUBR   A,S
        RMO    X,A
        ADDR   S,A
        RMO    A,S
        CLEAR T             # Preberemo f1.
        LDA    #3
        SUBR   A,T
        RMO    X,A
        ADDR   T,A
        STA    ADDR
        LDA    @ADDR
        CLEAR B             # Preberemo f2.
        LDT    #6
        SUBR   T,B
        RMO    X,T
        ADDR   B,T
        STT    ADDR
        LDT    @ADDR
        ADDR   T,A          # Seštejemo f1 in f2.
        STS    ADDR
        STA    @ADDR
        ...
        CLEAR S             # Začetek pogojnega stavka.
        LDA    #12          # if(f % 2 == 0)
        SUBR   A,S
        RMO    X,A
        ADDR   S,A
        STA    ADDR
        LDA    @ADDR        # Preberemo f.
        RMO    A,T
        LDA    #2
        RMO    A,S          # Izračunamo ostanek.
```

```
        RMO   T,A
        DIVR  S,A
        MULR  S,A
        RMO   T,S
        SUBR  A,S
        LDA   #0
        COMPR S,A          # Primerjamo števili.
        JEQ   L2
        J     L0           # Vrnitev na začetek zanke.
L2      ...                # Telo pogojnega stavka.
L3      CLEAR S            # Konec zanke.
        LDA   #9           # println(result);
        SUBR  A,S
        RMO   X,A
        ADDR  S,A
        STA   ADDR
        LDA   @ADDR
        STA   @SP          # Shranimo argument.
        JSUB  println
        LDA   #0           # Vrnemo vrednost 0.
        ...                # Epilog funkcije.
        RSUB
        END   Start
```

Izsek kode 5.12: Generirana zbirna koda testnega programa.

5.10 Objektne datoteke

V zadnji fazi zaledja prevajalnik s pomočjo zbirnika generirano zbirno kodo prevede v objektno datoteko, ki jo lahko izvedemo v simulatorju računalnika SIC/XE. Zbirnik, ki je del prevajalnika, lahko zaženemo tudi samostojno.

Prevajanje zbirnika se prav tako izvede v več fazah:

- **leksikalna in sintaksna analiza:** zbirnik prebere izvorni program, preveri ustreznost zbirnih ukazov in ustvari zaporedje vozlišč, ki vsebujejo podatke o ukazih in operandih,

- **razreševanje oznak in naslovov:** za vsako uporabljeno oznako in naslov izračunamo odmik oziroma absolutni naslov,
- **generiranje strojne kode:** vsak zbirni ukaz izvornega programa prevedemo v strojni ukaz,
- **generiranje objektne kode:** ustvarimo zaglavje in končni zapis objektne datoteke ter posamezne strojne ukaze združimo v programske zapise.

Zbirnik pri generiranju strojnih ukazov vsak zbirni ukaz ustrezno prevede v zaporedje znakov v šestnajstiškem sistemu oziroma sestavu. Strojni ukazi so v eni izmed štirih oblik ukazov računalnika SIC/XE, ki smo jih predstavili v četrtem poglavju.

Poleg tega moramo upoštevati tudi direktive zbirnega jezika, s katerimi določimo naslove in začetne vrednosti globalnih spremenljivk. Zbirnik vse strojne ukaze in podatkovna polja globalnih spremenljivk razporedi v enako zaporedje, kot se bodo naložili v pomnilnik računalnika SIC/XE.

V zadnji fazi zbirnik generira objektno kodo. Najprej ustvari zaglavje objektne datoteke, v katerega shrani ime programa, začetni naslov kode v pomnilniku in skupno dolžino kode. Nato za posamezne strojne ukaze in podatkovna polja ustvari ustrezne programske zapise. V vsak zapis shrani 60 bajtov oziroma največ 30 strojnih ukazov. Zatem vse zapise združi in jih zapiše v objektno datoteko.

S tem se celoten postopek prevajanja zaključi, prevedeni program pa lahko izvedeno v simulatorju računalnika SIC/XE.

Poglavje 6

Sklepne ugotovitve

V sklopu diplomskega dela smo se seznanili z arhitekturo računalnika SIC/XE in postopkom izdelave prevajalnika ter zbirnika.

V teoretičnem delu smo predstavili zgodovino in strukturo prevajalnikov, značilnosti programskega jezika C in arhitekturo računalnikov SIC ter SIC/XE, poleg tega pa smo si ogledali obliko objektnih datotek.

Izdelava prevajalnika je potekala v praktičnem delu, kjer smo se seznanili z leksikalnimi in sintaksnimi pravili programskega jezika C, postopkom razčlenjevanja izvorne datoteke in generiranjem vmesne ter zbirne kode.

Pri izdelavi prevajalnika smo se srečali z mnogimi težavami, ki so posledica omejenosti računalnika SIC/XE, kot je na primer pomanjkanje oblik ukazov in načinov naslavljanja. Te težave smo odpravili z različnimi triki na nivoju zbirne kode, vendar še vedno obstajajo nekatere omejitve, ki preprečujejo popolno podporo programskega jezika C.

Prevajalnik ima prostor za več izboljšav, saj bi lahko dodali podporo za računalnik SIC in večje število optimizacij. Prav tako bi lahko spremenili tudi sam pristop in podporo za računalnik SIC/XE dodali v prevajalnik GCC, kar bi omogočilo prevajanje veliko večjega števila programskih jezikov.

Izvorna koda prevajalnika in zbirnika je odprtokodna in je objavljena pod tretjo različico licence *GNU General Public License* [15]. Dostopna je na spletnem naslovu `https://github.com/KarboniteKream/scc/`.

Literatura

[1] L. L. Beck. System Software: An Introduction to Systems Programming (Third Edition). Pearson, 1996.

[2] A. Aho, J. Ullman, M. S. Lam, R. Sethi. Compilers: Principles, Techniques and Tools (Second Edition). Addison Wesley, 2006.

[3] A. W. Appel. Modern Compiler Implementation in Java (Second Edition). Cambridge University Press, 2002.

[4] B. W. Kernighan, D. M. Ritchie. The C Programming Language (Second Edition). Prentice Hall, 1988.

[5] First compiler of Grace Hopper. Dosegljivo: `http://history-computer.com/ModernComputer/Software/FirstCompiler.html`. [Dostopano 3. 8. 2015].

[6] GNU Compiler Collection (GCC) Internals. Dosegljivo: `https://gcc.gnu.org/onlinedocs/gccint`. [Dostopano 7. 7. 2015]

[7] ISO/IEC 9899:2011 Standard Draft. Dosegljivo: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`. [Dostopano 20. 7. 2015].

[8] IEEE floating point. Dosegljivo: `https://en.wikipedia.org/wiki/IEEE_floating_point`. [Dostopano 12. 8. 2015].

[9] ANSI C Yacc grammar. Dosegljivo:
http://www.quut.com/c/ANSI-C-grammar-y.html.
[Dostopano 20. 7. 2015].

[10] Parsing Expressions by Recursive Descent. Dosegljivo:
https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm.
[Dostopano 18. 7. 2015].

[11] What is BNF notation? Dosegljivo:
http://cui.unige.ch/db-research/Enseignement/analyseinfo/
AboutBNF.html. [Dostopano 22. 7. 2015].

[12] Introduction to Compilers. Lecture 33: Register Allocation. Dosegljivo:
http://www.cs.cornell.edu/courses/cs412/2008sp/lectures/
lec33.pdf. [Dostopano 26. 7. 2015].

[13] Register Allocation by Graph Coloring. Dosegljivo:
http://www.lighterra.com/papers/graphcoloring.
[Dostopano 26. 7. 2015].

[14] Even Fibonacci numbers. Dosegljivo:
https://projecteuler.net/problem=2. [Dostopano 9. 8. 2015].

[15] The GNU General Public Licence v3.0. Dosegljivo:
http://www.gnu.org/licenses/gpl-3.0.en.html.
[Dostopano 20. 8. 2015].